Interaction Between Fluid and Solid Body Surfaces in Fluid Simulation using Material-Point Method

Dody Dharma[1], Rinaldi Munir[2], Tito D. Kesumo Siregar[3]
Informatics Research Group
School of Electrical Engineering and Informatics
Institut Teknologi Bandung, Indonesia
[1]dody@stei.itb.ac.id, [2]rinaldi@stei.itb.ac.id, [3]13511018@std.stei.itb.ac.id

Abstract— **Recently, fluid simulation computation is not only limited for research or industrial purpose but can also be implemented into personal computer software. With the advent of interactive real-time fluid simulation. One challenge of real-time fluid simulation is to simulate interaction between fluids and solid bodies. In this paper, we extend a two-dimensional material-point method (MPM) based fluid simulation with fluid particle and solid body surface interaction calculation. To simulate the interactions, we use several geometry concepts such as reflection and shape in order to formulate the necessary equations of fluid particle velocity change. The equation is then implemented into an existing MPM-based fluid simulation. Based on the benchmark results, the proposed fluid-solid body interaction method is viable for real-time fluid simulation. Performance drop between 21%-26% is observed in the implementation, with the maximum number of particles to be simulated while maintaining the average frame rate above 30 FPS is 75,000 particles. Finally, we found that the number of particles and solid body complexity affects the fluid simulation performance, while the number of solid body polygons does not affect the fluid simulation performance.**

Keywords— fluid simulation, material point method, solid body, polygon, reflection

I. Introduction

Recently, fluid simulation is not only limited in usage for industrial or research purpose. It is possible to implement real-time fluid simulation in personal computer software, such as interactive applications or video games. This is because the capability of personal computer improves over time, while new and more efficient fluid simulation methods continues to be developed. For video games, it is possible to integrate fluids as part of the game mechanics in order to create more entertaining and realistic playing experience. One of the recent video games to use fluid simulation as an integral part of the game mechanism is *PixelJunk Shooter* [1].

Many fluid simulation methods have been developed using various models and approaches, such as smoothed-particle hydrodynamics (SPH), moving particle semi-implicit (MPS), and material-point method (MPM). MPM is a relatively new approach to fluid simulation, with possibilities to be developed further. Recent researches on fluid simulation using MPM are snow simulation in movie [2], interactive fluid simulation for mobile devices [3], and integration of graphics processing unit for fluid simulation computation [4].

One challenge in the development of real-time fluid simulation is to simulate interaction between fluids and other materials, such as solid bodies, in a relatively fast and inexpensive in terms of computing resources (memory and processing power). Several solutions have been proposed, such as using a coupling force to interfacing between fluid and solid body simulations [5] and performing direct velocity manipulation perpendicular to the solid body surface [6]. The latter solution has been successfully implemented for real-time fluid simulation using SPH, while no MPM-based real-time solution has been proposed so far. An ability to simulate fluid and solid body interaction would be useful for development of real-time MPM-based fluid simulation.

In this paper, we described a method to simulate fluid-solid body interaction in a MPM-based fluid simulation. This is done by performing additional calculation to the simulated fluid particles. In order to perform the calculations, we modelled the solid bodies into a data structure suitable for performing the calculations in linear time. To show that the proposed method is viable for use in real-time fluid simulation, we implemented the method by modifying an existing MPM fluid simulation code and performed benchmark with the implementation result.

II. Related Works

Interactive real-time fluid simulation based on smoothed-particle hydrodynamics (SPH) have been successfully developed previously [6]. Furthermore, an implementation of SPH for use in video game has been successfully developed and released [1]. SPH-based fluid simulations are particularly attractive since it is unnecessary to maintain a static grid, unlike other grid-based fluid simulations that requires an allocation of static grid to perform the necessary calculations. Furthermore, performing interaction between fluids and solid bodies is relatively easy because the position of fluid particles in SPH-based fluid simulations can be manipulated.

MPM-based fluid simulations have been implemented successfully for snow simulations in computer-animated 3D movie [2]. While MPM-based fluid simulations still model the fluid as particles, it still uses a Eulerian grid to calculate some variables such as force and pressure, making calculations of such variables relatively easier. Fluid particles also allow relatively simple fluid-solid interaction calculations by manipulating the position of the fluid particles. Finally, it is also trivial to implement a particle-based fluid renderer for MPM-based fluid simulations. Real-time MPM-based fluid simulations have been implemented in mobile devices [3], and the fluid simulation calculations have been successfully implemented as compute shader for calculations using graphics processing unit (GPU) instead of computer processing unit (CPU), greatly improving the simulation performance [4].

In order to perform fluid-solid interaction calculation, information about the solid body boundaries within the simulation is required. With the information of the boundaries, calculating the position of fluid particles is feasible by reflecting the particle velocity to the boundaries. Marching cubes algorithm has been used for visualizing the fluid in a fluid simulation implementation [6]. Using the algorithm, it is possible to obtain the boundaries of a solid body in a three-dimensional grid. The analogue of marching cubes algorithm in the two-dimensional grid is the marching squares algorithm.

We conclude that using a number of concepts such as geometrical reflections and marching squares algorithm, it should be feasible to provide a fluid-solid interaction method for a real-time MPM-based fluid simulation.

III. Related Concepts

In order to implement the interaction between fluid particles and solid body surfaces, a number of mathematical concepts related to geometry is studied, such as line equations and vector mathematics. We also use several algorithms and predicate tests related to computational geometry.

A. Line Equations

Generally, a line can be represented using the general linear equation in the form of $Ax + By + C = 0$. However, if two points of line is known, it is possible to represent the line in the form of $y - y_P = m(x - x_P)$, with (x_P, y_P) and (x_Q, y_Q) representing the points and $m = \frac{y_Q - y_P}{x_Q - x_P}$ is the line gradient. Care must be taken for straight vertical lines, where $x_Q - x_P = 0$.

Using the line equation in the form of $y = mx + c$, it is possible to determine whether a given point (x_i, y_i) is above or below the line. If $y_i > mx_i + c$, the point is located above the line; if $y_i < mx_i + c$, the point is located below the line.

B. Reflection

Reflection is the direction change of an object movement due to interaction with a flat surface. The concept can be applied to various objects, such as balls and light rays. Given an original movement velocity $\mathbf{v}$ and the reflected velocity $\mathbf{v}'$, relationship between the two vectors can be expressed as $\mathbf{v}' = \mathbf{v} - 2(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$, with $\hat{\mathbf{n}}$ denoting the unit normal vector perpendicular to the surface. The equation can be derived geometrically by observing the projection of v to the normal vector as shown in Fig. 1.

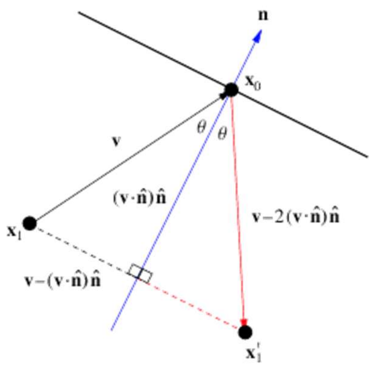

$\mathbf{n}$
$\mathbf{x}_0$
θ
θ
$\mathbf{v}$
$(\mathbf{v}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}$
$\mathbf{v}-2(\mathbf{v}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}$
$\mathbf{x}_1$
$\mathbf{v}-(\mathbf{v}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}$
$\mathbf{x}_1'$

Fig. 1 A diagram illustrating the reflection of a vector to a flat surface.

C. Counter-Clockwise Test

Counter-clockwise test is a useful predicate test in computational geometry to determine the ordering for a set of points in two-dimensional space. Given three points P, Q, and R, it is possible to determine whether $\overline{PQR}$ constitutes a "left-turn" (i.e. a counter-clockwise arc), a "right-turn" (i.e. a clockwise arc), or a straight line (i.e. Q is in $\overline{PR}$). The equation for the test can be derived from the cross-product result of two-dimensional vectors. For a set of three points, $\overline{PR} \times \overline{QR}$ results in a vector with three possibilities for the value of z-coordinate: positive, zero, or negative. A positive number implies that $\overline{PQR}$ form a "left-turn", a negative number implies a "right-turn", and a zero implies a straight line.

Generally, counter-clockwise test can be used to solve problems that require the set of points to be ordered based on certain angles. Counter-clockwise test is first used in an implementation of Graham scan algorithm to determine the convex hull from a set of points [7]. The counter-clockwise test can also be used to determine whether a line segment intersects each other [8].

D. Marching Squares Algorithm

Marching squares algorithm is an algorithm that can be used to determine a contour from a binary matrix. The algorithm works by determining the values of all 2×2 submatrices inside the matrix based on the value of each corner of the submatrices. For every submatrix, there are $2^4 = 16$ binary possibilities, each representing a contour solution. Fig. 2 shows all possible combinations for a single contour cell. The resulting matrix represents the contour of the original binary matrix, which can be used for a variety of applications. A real-time fluid simulation by Génevaux [5] used the marching cubes algorithm, which is the equivalent of marching squares algorithm in three-dimensional space, to visualize the particle fluids in a real-time fluid simulation.

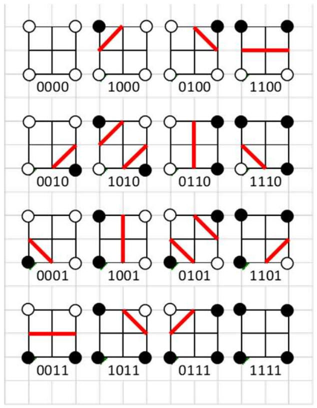


Fig. 2 All possible configurations of marching squares algorithm, with red lines denoting the resulting contour line.

IV. Design

A. Representation of Solid Bodies

In order to be able to perform the necessary calculations, we need to determine an appropriate model and data structure for the solid bodies in the simulation. Since the simulation is two-dimensional, the solid body should be in form of two-dimensional shapes.

For this simulation, we modelled solid bodies as polygons, represented as an array of points. Since a polygon is a surface constrained by a number of straight line segments and line segment can be represented by two points, we can represent a polygon as a set of points. We can argue that any shape with curved line segments can be approximated as a polygon using polygonal approximation algorithms, such as works by Zhu [9] and Kolesnikov [10].

A more useful model is to represent the solid bodies as a convex polygon. This is because we can also convert any polygon into a number of convex polygons using algorithms such as ear-clipping triangulation by Eberly [11]. Therefore, we can store the polygon points in counter clockwise order. This is required so that the test for testing whether a point is inside a polygon, as described in Fig. 3, can run in linear time.

B. Point Test for Convex Polygons

Given a list of polygon points ordered counter clockwise L and a point R, it is possible to determine whether R is inside or outside of the polygon. This is done by looping through every three consecutive points and perform the counterclockwise test. R is inside the polygon if and only if every three consecutive points of L formed a left-turn. Fig. 3 is the pseudocode of the described test as a Boolean function which returns true if R is inside the polygon of L and false otherwise.

```
function INSIDE(R: point, L: array of point): boolean
  for i in [1..n(L)]
    let j <- i + 1 if i < n(L), else 1
    let turn <- TURN($L_i$, $L_j$, R)
    if turn <> LEFT_TURN then
      <- false
  <- true
```

Fig. 3 Test to determine whether a point is inside or outside a polygon.

C. Polygon Rasterization

Given a set of polygons, we convert the polygons into a binary matrix. For every cell in the matrix, the value 1 represents a polygon and 0 represents an empty space. This can be done by looping through all matrix cells and checking whether the cell centre coordinate is inside a polygon.

A useful optimization can be done here by observing that the rasterization result remains the same if the set of polygons contains no changes. Therefore, the previously used binary matrix can be reused. In the implementation, we separate between static polygons (i.e. polygons that will not change during the simulation) and dynamic polygons (i.e. polygons that might change during the simulation). For the static polygon, we keep a binary matrix precomputed with the rasterization result. During the simulation, the binary matrix is simply copied and merged with the rasterization result of the dynamic polygons.

D. Contour Matrix

From the binary polygon matrix, the marching squares algorithm is used to determine the contour of every 2×2 submatrices. The result is a contour matrix, which contains an integer in the range between 0 (0000_2) and 15 (1111_2) in every cell.

E. Fluid Particle Velocity Calculation

Using the values from contour matrix, it is possible to determine whether a fluid particle is inside or outside a solid body. As illustrated in Fig. 2, each of the contour value represents a unique configuration consisting of a straight-line segment as boundary between the solid body area and the empty space. To determine whether a particle position is inside the solid body area, thus inside a solid body, we use the straight-line equations to determine whether the point is above or below the line. This information is enough to determine which the area of the particle belongs to. Care is taken for special cell cases, such as cell containing vertical lines, the horizontal lines, and cells containing multiple diagonals.

F. Integration into Fluid Simulation

Based on the material-point method fluid simulation steps in [3], it is suggested that the particle velocity calculation due to wall collision is done at later stages of the fluid simulation step. This is necessary to ensure that the fluid simulation is working with the correct particle position and velocity from the previous simulation step. On the other hand, the fluid particle velocity calculation will have to take into account particles located inside the solid body. Meanwhile, the polygon rasterization and contour matrix calculation do not depend on any previous values and can be done anytime.

Fig. 4 shows the proposed modified fluid simulation steps. The original fluid simulation steps are drawn with white background, while the inserted fluid-solid interaction steps are drawn with grey background.

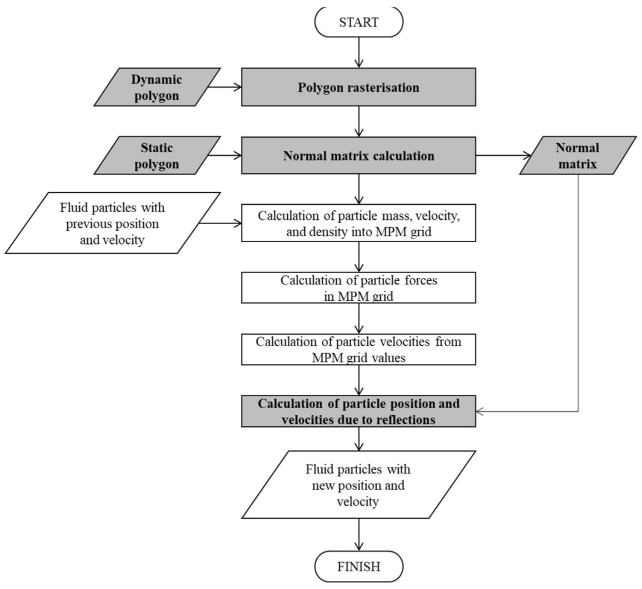

START
Dynamic polygon
Polygon rasterisation
Static polygon
Normal matrix calculation
Normal matrix
Fluid particles with previous position and velocity
Calculation of particle mass, velocity, and density into MPM grid
Calculation of particle forces in MPM grid
Calculation of particle velocities from MPM grid values
Calculation of particle position and velocities due to reflections
Fluid particles with new position and velocity
FINISH

Fig. 4 Flowchart of a single fluid simulation update steps with additional fluid-solid interaction steps inserted (shown in grey background).

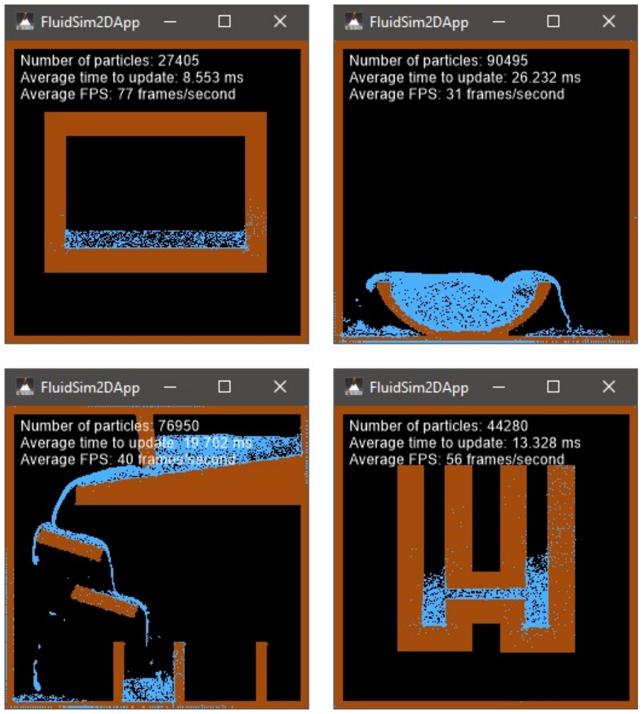
V. Implementation Results and Analysis

We implemented the proposed fluid simulation using Cinder programming library in C++ language. The library provides several application programming interfaces to perform drawing using OpenGL. Cinder also provides a built-in helper library for vector calculations.

To avoid reinventing the wheel, an existing fluid simulation implementation is modified instead of reimplement the whole fluid simulation code. This allows the research to focus on the fluid-solid interaction part of the fluid simulation.

A. Implementation Issues and Solutions

The implemented fluid simulation can generally produce visually correct fluid simulation with interaction between fluids and solid bodies. However, one particular bug in the implemented fluid simulation is the presence of fluid particles inside the solid bodies. This occurs because the particles are unable to exit the solid bodies if traversed too far inside the solid body grids. This is visible in parts of Fig. 5, where there are a number of particles (blue) trapped inside the solid bodies (brown). The problem is more visible with viscous fluids, which can attain greater velocity and traverse further inside the solid bodies. Another limitation of the implemented fluid simulation is the inability to represent very thin solid bodies, with most of the fluids will simply fall through. This happens if the fluid particles move fast enough to skip the solid body grids.



Number of particles: 27405
Average time to update: 8.553 ms
Average FPS: 77 frames/second

Number of particles: 90495
Average time to update: 26.232 ms
Average FPS: 31 frames/second

Number of particles: 76950
Average time to update: 19.702 ms
Average FPS: 40 frames/second

Number of particles: 44280
Average time to update: 13.328 ms
Average FPS: 56 frames/second

Fig. 5 Screenshots of the implemented fluid simulation program with various polygon shapes.

B. Implementation Performance

To determine whether the proposed fluid simulation method is viable for real-time fluid simulation, a number of benchmarks are performed using several test cases. The benchmark is done by running the fluid simulation with certain configuration for 20 seconds. Each second, three measurements are recorded into an external file: the average time to update (measured in milliseconds), the average time to draw (measured in milliseconds), and the average frame rate (measured in frames per second). The recorded measurement of first second is discarded since the first second also includes the time for simulation preparation, and the remaining recorded values are averaged. The benchmarks are performed using a computer with Intel Core i3-6006U processor, NVIDIA GeForce 940MX graphics chip, 8 GB DDR4 memory, and Windows 10 operating system.

1) Number of particles

The simulation is executed with increasing number of particles for each run, from 25,000 to 100,000. Table I shows the benchmark result, while Fig. 6 shows the graph of average update and draw time for each number of particles.

TABLE I BENCHMARK RESULT WITH INCREASING NUMBER OF PARTICLES

Number of Particles	Average *update* time (ms)	Average *draw* time (ms)	Average FPS
25,000	14.80	3.20	52.4
50,000	19.59	4.01	40.5
75,000	24.01	4.74	33.4
100,000	30.50	5.61	27.1

It can be observed that the time to update and draw increases linearly relative to the number of particles. This is expected from the fluid simulation, which has the algorithm complexity in order of $O(n)$ where n is the number of particles.

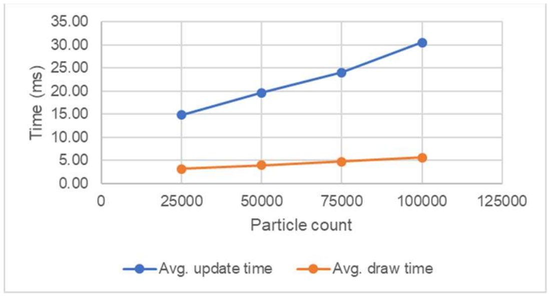


Fig. 6 Graph of average update and draw time with increasing number of particles.

2) *Number of polygons*

The simulation is executed with increasing number of polygons for each run, from 250 to 1,000. Table II shows the benchmark result, while Fig. 7 shows the graph of average update and draw time for each number of polygons.

TABLE II BENCHMARK RESULT WITH INCREASING NUMBER OF POLYGONS

Number of Polygons	Average *update* time (ms)	Average *draw* time (ms)	Average FPS
250	20.93	11.29	29.8
500	21.59	18.94	23.9
750	20.94	25.73	20.6
1,000	21.34	31.59	17.9

It can be observed that the number of polygons does not significantly affect the simulation performance, while the draw time increases linearly with the increasing number of polygons. This result shows that the time to draw polygons at screen dominates the time to perform simulation update for large number of polygons; simulation implementations should implement an efficient method to draw the polygons.

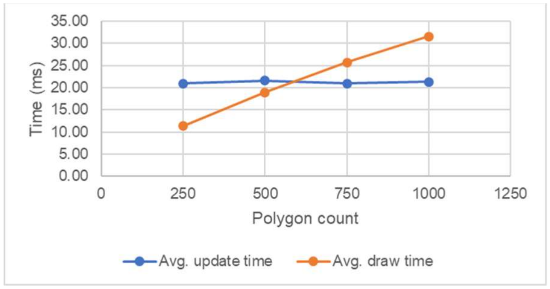


Fig. 7 Graph of average update and draw time with increasing number of polygons.

3) *Polygon complexity*

The simulation is executed with two different configurations, one containing a relatively simple solid body shape and the other one containing a relatively complex solid body shape with many curves and corners. Table III shows the benchmark result, while Fig. 8 shows the graph of average update and draw time for both simple and complex configurations.

TABLE III BENCHMARK RESULT WITH DIFFERENT POLYGON COMPLEXITY

Polygon complexity	Average *update* time (ms)	Average *draw* time (ms)	Average FPS
Simple	29.05	5.41	28.2
Complex	31.73	5.55	26.4

It can be observed that there is noticeable difference in performance between the fluid simulation with simple and complex solid body shape, with the simpler shape having a higher performance as expected. This result can be explained by the increased number of fluid-solid body interactions in complex simulation compared to the simple simulation, causing an increase in number of computations for the complex simulation.

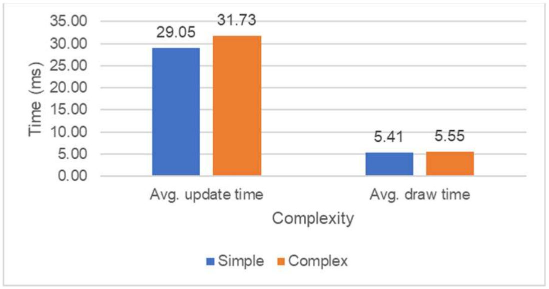


Fig. 8 Graph of average update and draw time with different polygon shape complexity.

4) *Performance drop*

For this benchmark, a special compilation of the fluid simulation program is built with the fluid-solid body interaction codes disabled via C++ pre-processor feature. Therefore, both programs run the exact same code except for the fluid-solid body interaction calculations. The benchmark is run with three different shapes of solid body: 'box' representing a simple boundary box, 'standard' representing a solid body with few number of corners and curves, and 'complex' representing a solid body with many corners and curves. Table IV shows the benchmark result, while Fig. 9 and Fig. 10 show the graph of average update and draw time of the simulation.

TABLE IV BENCHMARK RESULT WITHOUT AND WITH FLUID-SOLID INTERACTION

Measurement		Test case		
		box	standard	complex
Average *update* time (ms)	Without interaction	23.73	24.26	24.10
	With interaction	31.68	32.01	30.55
	Performance drop	25.08%	24.20%	21.12%
Average *draw* time (ms)	Without interaction	5.61	5.56	5.51
	With interaction	5.86	5.54	5.45
	Performance drop	4.24%	(0.47%)	(1.13%)
Average FPS	Without interaction	33.1	32.4	32.7
	With interaction	26.0	26.2	27.1
	Performance drop	27.08%	23.88%	20.51%

It can be observed that there is performance drop between fluid simulation without and with fluid-solid body interactions. This is expected from the implementation, since there are more calculations to be performed on the fluid simulation with fluid-solid interaction.

From the benchmark result, the performance drop ranges in 20.51%-27.08%, depends on the performance metric. This value can be used as performance lower bound; subsequent implementation of fluid simulations with fluid-solid interaction should be able to reach similar performance result.

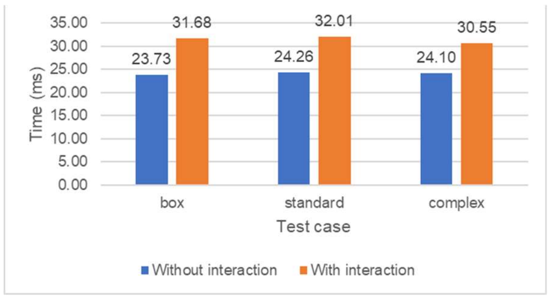


Fig. 9 Graph of average update time with various polygon shapes, comparing between simulation without and with fluid-solid interaction.

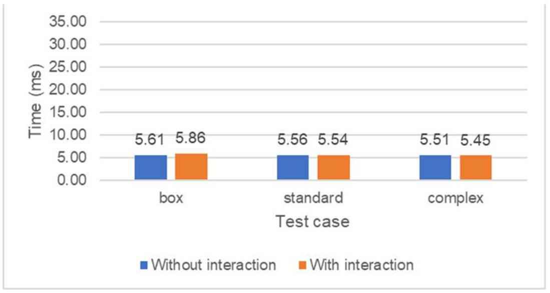


Fig. 10 Graph of average draw time for various polygon shapes, comparing between simulation without and with fluid-solid interaction.

VI. Conclusion

We successfully extend a two-dimensional material-point method-based fluid simulation with interaction between fluid particles and solid bodies. We modelled solid bodies as polygons, which is rasterized into a binary matrix. The marching squares algorithm is used to determine the contour of the binary matrix, which can be used to determine normal vectors in each cell of the matrix. Detecting whether a fluid particle collide with a solid body is done geometrically using the contour values, and particle velocity calculation is done using the determined normal vector.

To evaluate the proposed simulation method, the fluid simulation is implemented by modifying the existing fluid simulation implementation by Kotlin. Based on the benchmark result of the implementation, there is a 21%-26% performance drop between fluid simulation with fluid-solid interaction and without fluid-solid interaction. The maximum number of fluid particles that can be simulated while still keeping the average frame rate per second over 30 FPS is 75,000 particles. The factors affecting the fluid simulation performance is the number of particles and the complexity of the solid bodies. On the other hand, the number of polygons representing the solid bodies does not affect the fluid simulation performance.

References

[1] J. Kessler, "Go With the Flow! Fluid and Particle Physics in PixelJunk Shooter," *Game Developer Conference*, 2010. [Online]. Available: https://www.gdcvault.com/play/1012447/Go-With-the-Flow-Fluid. [Accessed: 31-Oct-2017].

[2] A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle, "A material point method for snow simulation," *ACM Trans. Graph.*, vol. 32, no. 4, p. 1, 2013.

[3] D. Dharma and A. Manaf, "Interactive fluid simulation based on material point method for mobile devices," *Adv. Informatics Concepts, Theory Appl. (ICAICTA), 2015 2nd Int. Conf.*, pp. 1–6, 2015.

[4] D. Dharma, C. Jonathan, A. I. Kistidjantoro, and A. Manaf, "Material point method based fluid simulation on GPU using compute shader," *2017 Int. Conf. Adv. Informatics, Concepts, Theory, Appl.*, no. August, pp. 1–6, 2017.

[5] O. Génevaux, A. Habibi, and J.-M. Dischler, "Simulating Fluid-Solid Interaction," *Proc. Graph. Interface 2003 Halifax, Nov. Scotia, Canada, 11 - 13 June 2003, 31-38*, pp. 31–38, 2003.

[6] M. Müller, D. Charypar, and M. Gross, "Particle-Based Fluid Simulation for Interactive Applications," *Proc. 2003 ACM SIGGRAPH/Eurographics Symp. Comput. Animat.*, no. 5, pp. 154–159, 2003.

[7] R. Sedgewick and K. Wayne, *Algorithms*. Addison-Wesley Professional, 2011.

[8] S. Halim, *Competitive Programming 2*. 2010.

[9] Y. Zhu, "Optimal polygonal approximation of Digitized curves," *IEE Proc -Vis Image Signal Process*, vol. 144, no. I, 1997.

[10] A. Kolesnikov, *Efficient algorithms for vectorization and polygonal approximation*. University of Joensuu, 2003.

[11] D. Eberly, "Triangulation by ear clipping," *Magic Software, Inc*, pp. 1–13, 2002.